Journal of Signal Processing Systems (2021) 93:513–529
https://doi.org/10.1007/s11265-021-01651-5

A Parametrizable High-Level Synthesis Library for Accelerating Neural Networks on FPGAs

Lester Kalms[1] · Pedram Amini Rad[1] · Muhammad Ali[1] · Arsany Iskander[2] · Diana Göhringer[1]

Received: 4 May 2020 / Revised: 4 November 2020 / Accepted: 10 February 2021 / Published online: 15 March 2021
© The Author(s) 2021

Abstract

In recent years, Convolutional Neural Network CNN have been incorporated in a large number of applications, including multimedia retrieval and image classification. However, CNN based algorithms are computationally and resource intensive and therefore difficult to be used in embedded systems. FPGA based accelerators are becoming more and more popular in research and industry due to their flexibility and energy efficiency. However, the available resources and the size of the on-chip memory can limit the performance of the FPGA accelerator for CNN. This work proposes an High-Level Synthesis HLS library for CNN algorithms. It contains seven different streaming-capable CNN (plus two conversion) functions for creating large neural networks with deep pipelines. The different functions have many parameter settings (e.g. for resolution, feature maps, data types, kernel size, parallelilization, accuracy, etc.), which also enable compile-time optimizations. Our functions are integrated into the HiFlipVX library, which is an open source HLS FPGA library for image processing and object detection. This offers the possibility to implement different types of computer vision applications with one library. Due to the various configuration and parallelization possibilities of the library functions, it is possible to implement a high-performance, scalable and resource-efficient system, as our evaluation of the MobileNets algorithm shows.

Keywords High-level synthesis · Neural networks · FPGA · Hardware acceleration · Library

1 Introduction

Nowadays neural network applications are widely used in new technologies such as artificial intelligence and robotics [23]. Convolutional Neural Network (CNNs) [12] are a type of deep neural networks, which are significantly successful in object detection [25] and image classification [21] due to the ability to extract both spatial and temporal features [12]. However, the performance relies on the computing platform and implementation. The first challenge in accelerating CNNs is that they have a huge memory requirement, since common CNN models use millions of trained parameters. Furthermore, CNNs are computationally intensive with over billions of operations for the inference. Due to these challenges, GPUs [18], ASICs [4] and FPGAs [30] are mainly used for accelerating the CNN inference. Because of the advantages of high performance, energy efficiency and flexibility, FPGAs are attracting the attention of researchers to accelerate CNNs [27]. On the other hand, a straight forward design for FPGAs written in VHDL or Verilog can achieve a suitable performance. However, an effective and precise hardware design requires a high time to market and a lot of effort. Moreover, a flexible development framework like Caffe [13] and TensorFlow [1] for CPU and GPU is not available for FPGAs. To address this, High-Level Synthesis (HLS) tools from FPGA vendors, such as Xilinx's Vivado HLS [34] or Intel's OpenCL SDK [10], reduce the programming difficulty and shorten the development time remarkably, making FPGA-based solutions more popular.

✉ Lester Kalms
lester.kalms@tu-dresden.de

Pedram Amini Rad
pedram.amini_rad@tu-dresden.de

Muhammad Ali
muhammad.ali@tu-dresden.de

Arsany Iskander
arsany.eskander@student.guc.edu.eg

Diana Göhringer
diana.goehringer@tu-dresden.de

[1] Technische Universität Dresden, Dresden, Germany

[2] German University in Cairo, New Cairo, Egypt

Our contribution consists of a generic, template-based and open source HLS library for a fast implementation of CNNs on FPGA-based embedded or HPC systems. The library consists of 7 different layers, which are used in common CNN algorithms. It operates on parameterizable fixed-point data types and floating-point data types, and has been optimized for performance and resource efficiency. The different compile time parameters and data types of the library functions offer multiple opportunities for an optimized design and extensive design space exploration. One benefit is that all functions are streaming capable to allow a deep pipeline. Creating streaming applications with multiple nodes or layers gives FPGAs the ability to achieve higher performance and power efficiency for computer vision algorithms compared to other architectures, such as CPUs and GPUs, as Kalms et al. [14] or Qasaimeh et al. [24] show. Furthermore, we have researched and implemented different possibilities of parallelization in order to achieve a high performance with an efficient use of resources, which we show in this paper using our implementation of the MobileNets algorithm [9]. Our library is integrated into the HiFlipVX library, which is an open source HLS FPGA library for image processing [17] and object detection [15]. This offers the possibility to design and implement different kinds of computer vision applications with one library. Most functions of the libraries are based on the OpenVX standard. This simplifies the design of applications on heterogeneous systems containing different types of architectures (e.g. CPU, GPU and FPGA), due to the different existing implementations from different vendors.

In the following, Section 2 provides information about the related work, Section 3 describes the implementation of the neural network library and MobileNets, Section 4 evaluates the achieved results and Section 5 contains conclusion and outlook.

2 Related Work

State-of-the-art CNN architectures for large-scale visual recognition use a multitude of layers with millions of computations. FPGA designers for embedded applications encountered three major challenges to efficiently map CNNs on hardware such as a difficult programming framework, limited FPGA resources and memory bandwidth. Many implementations have been proposed to address the above mentioned challenges on FPGAs. Wang et al. [33] built an RTL library to map neural networks on FPGAs. However, RTL implementations suffer from high costs and time-to-market which makes RTL-based custom hardware accelerators infeasible for most cases.

The availability of HLS tools, using OpenCL, C or C++, from FPGA vendors (e.g. Vivado HLS [34] from Xilinx, SDSoC [26] from Xilinx or the OpenCL SDK [10] from Intel) reduces the programming hurdle and shortens the development time of FPGA-based hardware accelerators. Consequently, many HLS implementations have been introduced for the acceleration from CNNs, like from Tapiador et al. [30], Zhang et al. [37] or Venieris et al. [32], to implement energy-efficient and effective HLS-based neural network accelerators. While some papers present approaches for cloud applications with sufficient resources [3], others present designs for embedded applications with limited resources [37]. For example, Yao et al. [3] implemented an HLS-based library for cloud systems, like the AWS. To address the resource limitation on FPGAs, many optimizations and implementations were carried out to reduce the resource usage. Suda et al. [28] propose an implementation using HLS for a lighter data type (fixed-point 16-bit) while our proposed work supports multiple data types (32-bit floating-point and 8-bit, 16-bit or 32-bit fixed-point with an adjustable size of the fraction part).

Guo et al. [7] proposed a flexible CNN accelerator with bit-width reduction using quantization, improving the performance of OpenCL-based FPGA accelerators for CNNs. Liu et al. [19] integrated the pointwise separable convolution, which is needed in different neural networks like MobileNets. Some of the previous studies focused only on the acceleration of the convolution layers of CNNs. For example, Liu et al. [20] only used models with convolution layers without any Fully Connected layer. Therefore, it is hard to be used for accelerating different CNN algorithms.

Memory bandwidth issues in CNNs are discussed by Zhang et al. [37] and Zhang et al. [39]. Guan et al. [6] proposed FP-DNN, which is an end-to-end framework that automatically generates optimized FPGA-based implementations of deep neural networks (DNNs) using an RTL/HLS hybrid library. Another HLS based library is the Caffeine FPGA engine [38] that uses an HLS-based systolic-like architecture to implement matrix multiplication kernels. It allows changing parameters such as the number of (PEs), precision, and feature map size.

The proposed CNN library is highly parametrizable, has a rich set of functions and is therefore applicable for various algorithm designs. All functions are streaming capable and can be easily connected to each other. High performance with an efficient use of resources can be achieved through the streaming approach and the various parallelization parameters. The integration of the proposed CNN library into the HiFlipVX image processing library [17], which has been extended for object detection [15], increases the range of possible applications that can be implemented in the field of computer vision. Following the OpenVX standard [5] makes it easier to create a heterogeneous system consisting of different architectures (e.g. CPU, GPU and FPGA) from different vendors. Since the library does not require

vendor-specific or other external libraries, it can be ported to other platforms more easily. This also improves the verification and integration process in frameworks like Tensorflow [1] or Caffe [13].

3 Implementation

This section first describes the architecture and implementation of the different Neural Network layers. The library contains 7 neural network functions, which are described in Section 3.2. All functions are streaming-capable to exploit the advantages of an FPGA. Section 3.3 describes how to create an algorithm using the library components by using MobileNets as an example. It also adds two additional functions needed to create an efficient implementation.

3.1 The HiFlipVX Library

HiFlipVX is an open source [16] HLS FPGA library for image processing [17], which has been extended for object detection [15]. The library contains 46 C++ streaming-capable functions, which are mostly based on the OpenVX standard. OpenVX is an open, royalty-free standard for cross platform acceleration of computer vision applications [5]. The library functions are parametrizable using C++ templates and highly optimized for performance and FPGA resources. In comparison to the xfOpenCV library from Xilinx [36], it only consumed in average 39% FFs and 32% Lookup Table (LUTs) for a selected set of functions [17]. In addition to the OpenVX standard, most functions support different vectorization options (2x, 4x, 8x) and additional data types (8-, 16- or 32-bit signed/unsigned integers). The use of vectorization does not only increases performance, but also the energy efficiency, as shown by Akguen et al. [2].

The functions of our proposed library were integrated into the HiFlipVX library. They use the same data types and the function headers have a similar structure. Therefore it is easy to connect the functions of the two libraries, either directly or e.g. by using data-width converters. Furthermore, certain pre-processing for the CNNs can be done with the functions of the HiFlipVX library, e.g. changing the image size or the image format. The integration also makes it easier to use existing OpenVX-based frameworks, which did not observe CNNs, like AFFIX [29] or JANUS [22].

3.2 Neural Network Layers

One goal of the library was the streaming capability of the library functions. Since all functions are pipelined, a pipeline interval of 1 was a key objective to achieve an optimized performance. Additionally, all functions in the library have integrated vectorization, which can be applied on their Input Feature Map (IFM) and/or their Output Feature Map (OFM). Unsigned and signed 8-bit and 16-bit fixed-point and 32-bit floating-point data types are possible for the inputs, outputs, weights and biases, to be applicable for many hardware designs. The size of the fraction can be configured as a parameter of the function. For an 8-bit unsigned integer data type, this value can be between 0 and 8. If the *fixed-point position* is set to 5, the fractional part is 5 and the integer part is 3. Functions that need trained coefficients buffer them on first use, if configured, to reduce the amount of global memory access. The fixed-point implementations contain policies for rounding and overflow. If an overflow occurs data can either be truncated or saturated to its maximum/minimum value. For fixed-point arithmetic operations, the data can be rounded to zero or the nearest number.

Seven different neural network layers were designed and implemented: 3D Convolution, Depthwise Convolution, Pooling, Activation, Batch Normalization, Fully Connected and Softmax. The I/Os of the different layer functions are the input vector, the output vector and, if required, the weights vector and the biases vector. Since all functions are streaming capable, we can use the simple AXI4-Stream interface for the Xilinx implementation. It is a simple protocol, with ready and valid signal for handshaking and the corresponding data signal. The `HLS interface axis` directive in the library functions automatically creates this interface. For all interface parameters we use the `vx_image_data<DATA_TYPE, VEC_SIZE>` of the HiFlipVX library. It is a vector data type, with two additional configurable signals for the AXI4-Stream interface that can be activated by using macros. These signals indicate the Start of Frame (SoF) (`last` signal) and End of Frame (EoF) (`user` signal) and are needed when connected to the DMA or Video DMA (VDMA) blocks from Xilinx. The remaining library function parameters are template parameters (e.g. input/output image size, kernel size, IFM, OFM, etc.).

The library has been optimized and tested for Vivado HLS [34] and SDSoC [26] 2019.1, but also works with other versions. Internally SDSoC uses HLS, but builds a complete system around the accelerated functions containing the hardware and software layers. To create such a system, SDSoC adds some restrictions that basically affect the interfaces of the function. One of these limitations is that only `structs` with more than 1 element are synthesizable. This has been solved by automatically using native data types instead of *structs* for these kind of interfaces. This is also possible, since SDSoC adds the SoF and EoF signals to the AXI4-Stream interface by itself. Furthermore, interface arrays need a known amount of elements. The

proposed library does not need vendor specific or external libraries. For some mathematical operations or signals, Xilinx libraries have been used for a resource efficient implementation. By using a macro, alternatives are applied if other tools are used.

3.2.1 3D Convolution

The process of 3D convolution is the most computational intensive layer in most feed forward networks. The main goal of the proposed image and loop dimension ordering was to achieve a streaming capable function. Under this constraint we developed a structure that is optimized for performance and resource usage. Therefore, the ordering of some dimensions is different from the OpenVX standard. Listing 1 shows the general structure of the hardware implementation, which is be explained throughout this subsection. Therefore the total latency can be derived from the total number of loop iterations plus the pipeline stages. The order of the image and coefficient dimensions is:

- Input Image: $BATCH \times ROW_{src} \times COL_{src} \times IFM$
- Output Image: $BATCH \times ROW_{dst} \times COL_{dst} \times OFM$
- Weights: $OFM \times IFM \times K_y \times K_x$
- Biases: $(0) \vee (OFM) \vee (BATCH \times ROW_{dst} \times COL_{dst} \times OFM)$

As shown, different sizes for the Bias are possible. A stride is set, when input and output resolution differ. In the proposed implementation the stride only effects the condition when a result is written to the output. It has no effect to the latency. However, loop iterations could also be skipped in dependence of the stride. However, the used HLS compiler only allows "perfect loops", which could

```
1  for (b = 0) to (BATCH − 1)
2    for (y = 0) to (ROW_src + ⌊K_y/2⌋ − 1)
3      for (x = 0) to (COL_src + ⌊K_x/2⌋ − 1)
4        for (o = 0) to (OFM/V_o − 1)
5          for (i = 0) to (IFM/V_i − 1)
6            ReadInputVector()
7            UpdateSlidingWindow()
8            UpdateBuffers()
9            ComputeConvolution()
10           WriteOutputVector()
```

Listing 1 General structure of the 3D convolution function. Input Image Resolution: $(ROW_{src} \times COL_{src})$; Output Feature Map: (OFM); Input Feature Map: (IFM); Output Vectorization: (V_o); Input Vectorization: (V_i); Kernel Size: $(K_y \times K_x)$

be a drawback of using HLS. The general equation for calculating a 3D convolution is:

$$dst_{y,x,o} = \sum_{i=0}^{IFM-1} \sum_{n=0}^{K_y-1} \sum_{m=0}^{K_x-1} \times \left(src_{(y+n-\frac{K_y}{2}),(x+m-\frac{K_x}{2}),i} \cdot W_{o,i,n,m} \right) + B_o \quad (1)$$

Parallelization The performance benchmark for most 3D convolution layers is the number of multiplications processed per second. For a multiplication mostly internal Digital Signal Processor (DSPs) are used on an FPGA. When increasing the number of multiplications, the amount of data that is needed simultaneously and thus the required memory bandwidth increases. To implement an efficient streaming capable function, data of the input image as well as the coefficients should be buffered. This can limit the maximum resolution of the image to be processed. These buffers are usually implemented using Block RAM (BRAM). However, BRAM has a limited bandwidth to read and write data. To increase the bandwidth, data can be distributed over several BRAMs. However, this can lead to fragmentation, if the BRAM is not fully utilized and can therefore limit the data to be stored. For this reason, fragmentation should be kept as small as possible while increasing the amount of multiplications.

Various loop variables are suitable for parallelization, as illustrated in Listing 1. One possibility of parallelization would be in the direction of the (COL) as in HiFlipVX. However, this type of parallelization would increase the bit-width of various buffers and therefore lead to a high fragmentation of BRAM. Additional buffers would also have to be introduced to restructure the input and output data. Therefore, we have concentrated on the parallelization of the inner loops, as shown by the parameters (V_o) and (V_i) in Listing 1. Both (OFM) and (IFM) parallelization would increase the bit-width of the coefficient buffer. Additionally, the parallelization of (IFM) increases the bit-width of the input buffers. In some cases (V_i) can be raised to a certain point without causing additional fragmentation of the input buffer.

Structure of Buffers Figure 1 shows the structure needed to buffer the input data to achieve the sliding window effect for the 3D convolution function. It also shows the size of the different buffers, all of which have a depth of (V_i) elements. Additionally, the image shows the read and write operations between the different blocks by the dashed and dotted lines. The line buffers store complete rows of the image including all feature maps $(COL_{src} \cdot \frac{IFM}{V_i})$. Its height of $((K_y - 1) \cdot V_i)$ elements is stored as 1 element in the BRAM to reduce BRAM usage, since it can reduce

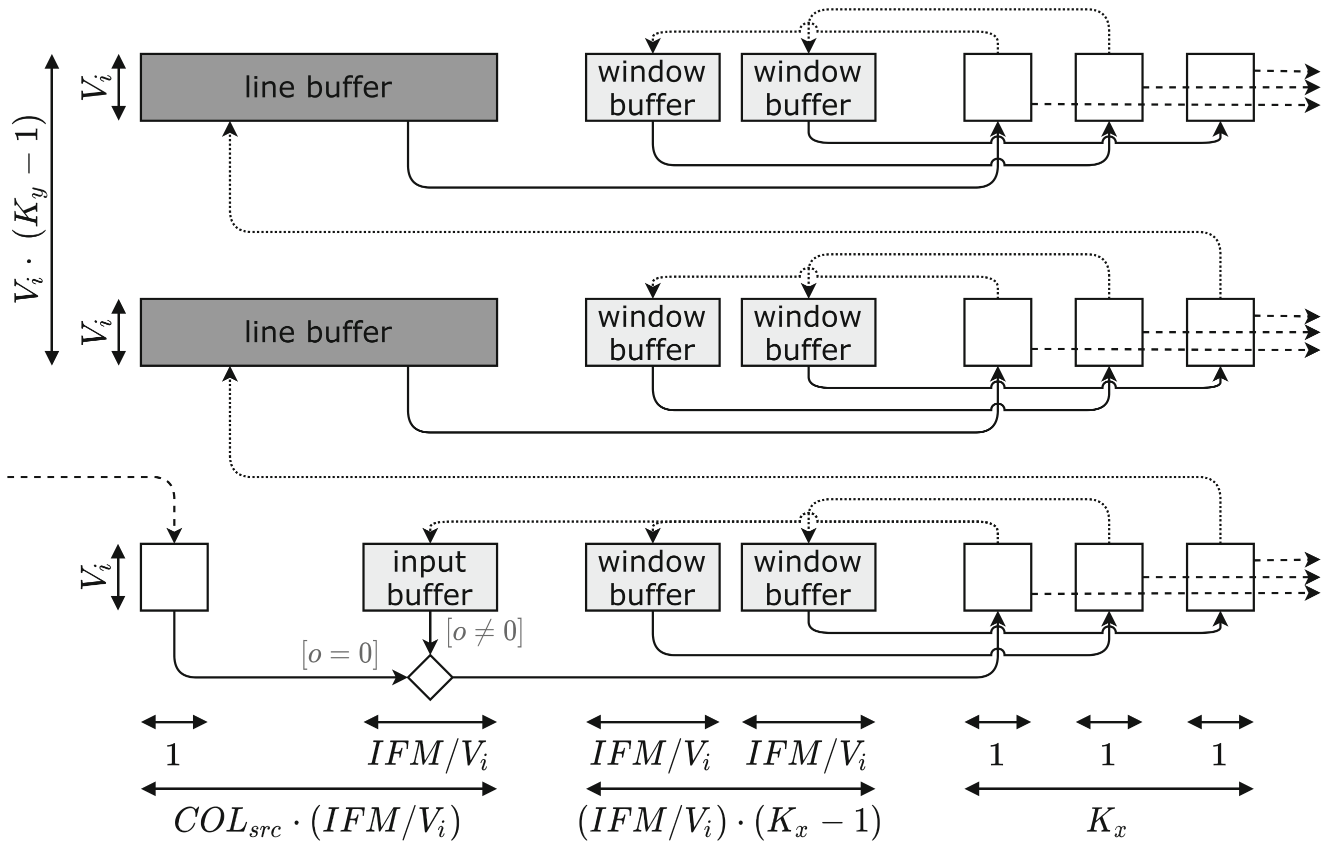

Figure 1 The input stage buffers the input image for the 3D convolution function with a $K_y \times K_x$ window/kernel size (here 3×3) and an input vector size of V_i. The input stage contains input registers on the left (white), big line buffers (dark gray), small input/window buffers (light gray) and sliding window registers on the right (white). The process of buffering the input image can be separated into 4 pipelined steps. 1. step reads new input vector of size V_i *(dashed lines)* while computing the first Output Feature Map $[o = 0]$ 2. step updates window registers *(continuous lines)*. 3. step updates buffers *(dotted lines)*. 4. step sends all data from the sliding window registers to the compute stage *(dashed lines)*. IFM = Input Feature Map; COL_{src} = Input Image Columns.

fragmentation. The input buffer is a line buffer that does not have to store the entire image row. Instead, it is sufficient to store the $(\frac{IFM}{V_i})$ elements of the current iteration of (x). The sliding window updates its complete elements in each clock cycle, because all feature maps of (x) are calculated before the window is moved one element to the right. The window buffers are needed for this, since only 1 element can be read from each line/input buffer in a clock cycle. Each of the $(K_y \cdot (K_x - 1))$ window buffers has $(\frac{IFM}{V_i})$ elements. The different computation steps of the 3D convolution are described below in chronological order.

1) Read Input Vector: Reads vector of V_i elements from the input image, if the following condition is met: $(y \leq ROW_{src}) \wedge (x \leq COL_{src}) \wedge (o = 0)$.

2) Update Sliding Window: In this stage, the data is read from the different buffers and stored in the sliding window, as shown in Figure 1 by the dashed lines. Each element in the sliding window of size $(K_y \times K_x)$ contains (V_i) vector elements. The left column of the sliding window get its data from the line buffers and the input buffer. If $(o = 0)$ new data is read from the input image instead of the input buffer. The other elements of the sliding window get their data from the window buffers. Additionally, the algorithm checks whether valid data should be present in the buffers. Otherwise a zero is loaded into the corresponding sliding window elements, to apply zero padding. The proposed implementation always applies zero padding of $(\lfloor\frac{K_x}{2}\rfloor)$ on both sides in x-direction and of $(\lfloor\frac{K_y}{2}\rfloor)$ on both sides in y-direction.

3) Update Buffers: This stage reads the data from the window and writes it to the different buffers, as shown in Figure 1 by the dotted lines. The input buffer receives its data from the bottom left element in the window. Since the input data can only be read once, it must be buffered. The line buffer receives its data from the right column of the window in the last iteration of $(o = \frac{OFM}{V_o} - 1)$. This moves the data of the image one line up so that it is available again at the next iteration of (x). The window buffer receives its data from the left columns of the window in the last iteration of $(o = \frac{OFM}{V_o} - 1)$. The sliding window effect results from the offset reading and writing between the window buffer and the window.

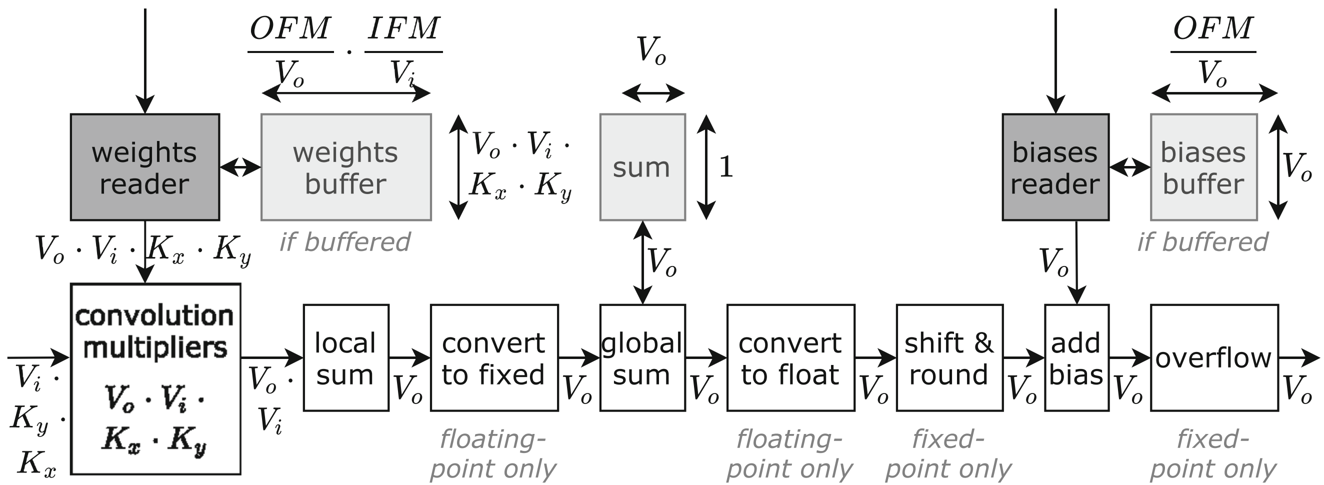

Figure 2 Computation stages of the 3D convolution implementation. The input comes from the sliding window of the input stage (Figure 1). Some stages are for floating-point or fixed-point numbers only. Buffers for weights/biases are marked in light gray. Reader functions (dark gray) buffer weights/biases if configured. IFM = Input Feature Map; OFM = Output Feature Map; V_i = Input Vector Size; V_o = Output Vector Size; $K_x \times K_y$ = Kernel Size.

4) Compute Convolution: Figure 2 shows the computation stage of the 3D convolution process. As stated, some of the blocks in the image are only used for fixed-point or floating-point calculations. The gray blocks show the data whose contents needs to be maintained between loop iterations and stored in buffers. The weight and bias coefficients can be buffered within the function if the user sets the appropriate parameter. On first use, they are read from the interfaces and stored in the buffers. If the same coefficients are needed again, they can be accessed from the buffers.

In the first step, the input data is taken from the sliding window and multiplied by the corresponding weights. In total $(V_o \times V_i)$ 2D-convolutions of the size $(K_y \times K_x)$ are calculated. Then (V_i) 2D-convolutions of the different (V_o) are added together to partially calculate the 3D convolution of each (V_o).

The operation of calculating a sum over several loop iterations violated the desired pipeline interval of 1 by a factor of 5 when using floating-point numbers with the Xilinx tools. Therefore, we convert floating-point numbers for this summation to a value that is saturated to a 32-bit fixed-point number. The user sets the parameter for the fixed-point position of this variable. In the next step the partial 3D convolutions are added to the final 3D convolution until all 2D-convolutions are summed up. Then the result is converted back if the final output should be a floating-point number.

When using fixed-point numbers, the multiplication in the 2D-convolution increases the fixed-point position. Therefore, the value is shifted back to the fixed-point position, while ensuring the overflow policy. This process is done before adding the bias, because it has the same fixed-point position as the output. After adding the bias, the result is checked for overflow and saturated if the corresponding policy is set.

4) Write Output Vector: Writes back a vector of V_o elements to the output image, if the following condition is met: $\left((y - \lfloor \frac{K_y}{2} \rfloor) \bmod (\frac{ROW_{src}-1}{ROW_{dst}-1})\right) \wedge \left((x - \lfloor \frac{K_x}{2} \rfloor) \bmod (\frac{COL_{src}-1}{COL_{dst}-1})\right) \wedge \left(i = \left(\frac{IFM}{V_i} - 1\right)\right)$. The condition includes the stride computation, expressed with the modulus operation. The value for the stride must be an element of the natural numbers.

3.2.2 Depthwise Convolution

The *Depthwise Convolution* can be considered as a 2D convolution that is applied to each feature maps of a 3D input image separately. This layer is usually used together with a "pointwise" convolution of (1 × 1), as in MobileNets [9]. This means that for a (3 × 3) convolution, a (3 × 3) pointwise convolution and a (1 × 1) pointwise convolution is used. The advantage of this approach is that less multiplications and weights are required for the convolution process. Comparing it to a classic 2D convolution, it has a similar effect to the separable filter shown in [17].

The amount of feature maps in the input and output image are the same for this function. When comparing with the structure of Listing 1, the loop over OFM is eliminated. Consequently, the total latency is reduced by that factor and there is only one parallelization term (V_i). The rest of the basic structure in Listing 1 remains. The total number of multiplications and weights is reduced by a factor of OFM compared to a pointwise convolution. Therefore fewer weights must be stored in the internal buffers. On the other hand, the size of biases remains unchanged. It

is still possible to choose between the different sizes of biases. Furthermore, the stride computation remains the same.

Compared to the structure in Figure 1 nothing changes for the buffering of the input image and the sliding window. As pointwise convolution only performs 2D convolution operations, Figure 2 omits the summation blocks. This eliminates the need for inter-loop summation and the conversions for floating-point numbers. Except for the amount of weights and convolutions, the rest of the structure in Figure 2 remains. The conditions when a vector is read from the input image or when it is written to the output image only change in such a way that the following conditions are omitted: $(o = 0)$ for the input and $(i = (\frac{IFM}{V_i} - 1))$ for the output. This also implies that the stride calculation remains the same.

3.2.3 Pooling

The purpose of the *Pooling* layer is to reduce the spatial size of the image to reduce the number of parameters and calculations in the neural network. The pooling operation works independently on each feature map. Similar to a 2D convolution a window slides over an input image. To calculate the output, the values in the window are either averaged or the maximum value is taken. With the help of a stride, pixels can be skipped so that the output image becomes smaller than the input image. The following equation is used to calculate the stride in x direction:

$$Stride_x = \left\lfloor \frac{COL_{src} + 2 \cdot Pad_x - K_x}{COL_{dst}} \right\rfloor \tag{2}$$

The window $(K_x \times K_y)$ can have any size between (1×1) and $(ROW \times COL)$. Like in the convolution filters zero padding can be applied. The padding size $(Pad_{x,y})$ is in the range between (0) and $\left\lfloor \frac{K_{x,y}}{2} \right\rfloor$. The calculated $Stride_{x,y}$ is in the range between (1) and $(K_{x,y})$. The overall structure of the function is very similar to the pointwise convolution, without the need of buffering coefficients. The total latency only differs slightly, since the padding size is not fixed: $((ROW_{src} + Pad_y) \times (COL_{src} + Pad_x) \times \frac{IFM}{V_i})$. For average pooling the *sum* of all window elements is calculated and then multiplied by the normalization. For fixed-point values an additional operation is needed that shifts the result back to the desired fixed-point position (FP).

$$avg = \lfloor \lfloor sum \cdot \underbrace{\left\lfloor \frac{2^{FP}}{K_y \cdot K_x} \right\rfloor}_{normalization} \rfloor \underbrace{\cdot 2^{-FP}}_{shifting} \rfloor \tag{3}$$

3.2.4 Activation

The *Activation* function is a crucial component in CNNs. In general, the function is connected to each neuron in the network and determines whether it should be activated or not. Table 1 shows the 9 implemented activation functions, which have been defined in the OpenVX standard. For fixed-point numbers the overflow policy needs to be applied to the following functions: $softrelu$, $square$ and $linear$. In addition, an overflow can occur when calculating the absolute function for a signed data type and if the value is the smallest possible. For fixed-point numbers the rounding policy must to be applied to the following functions: $logistic$, $softrelu$, $square$ and $linear$. The logarithmic and exponential activation functions are computed using floating-point operations, due to the high range of possible values and the resulting accuracy loss when using fixed-point numbers. Therefore, conversions are needed for fixed-point input and output images using multiplication operations. As shown in the table, the hyperbolic tangent function is calculated with 1 repeated exponential function and 1 division to the reduce resource usage. The most resource efficient fixed-point square root was the one from the Xilinx HLS library. It is automatically selected, when using Xilinx tools, otherwise HiFlipVX proposed functions is used [17]. The activation function can be computed in parallel (V) in a SIMD manner on the 3D input image. The latency of the hardware function is: $ROWS \cdot COLS \cdot \frac{IFM}{V} + P$.

3.2.5 Batch Normalization

Batch Normalization is a technique to improve the stability and performance in neural networks. The core idea is that the inputs of each layer of an image are normalized so that the mean output activation is zero and the standard deviation is one [11]. The Batch Normalization calculates a mini-batch (B) over a set of pixels values (x_i): $B = \{x_1, x_2, ..., x_{IFM}\}$. Considering a three dimensional input image $(src_{x,y,i})$, the mini-batch would be calculated over

Table 1 Implemented activation functions.

logistic	$f(x) = \frac{1}{e^{-x}}$
hyperbolic tangent	$f(x) = a \cdot tanh(b \cdot x) = a \cdot \frac{e^{2 \cdot b \cdot x} - 1}{e^{2 \cdot b \cdot x} + 1}$
relu	$f(x) = max(0, x)$
bounded relu	$f(x) = min(a, max(0, x))$
soft relu	$f(x) = log(1 + e^x)$
abs	$f(x) = \lvert x \rvert$
square	$f(x) = x^2$
square root	$f(x) = \sqrt{x}$
linear	$f(x) = a \cdot x + b$

the third dimension of size IFM. It first calculates the mean (μ) of the pixel values, as shown in Eq. 4. Using the the mean value, the variance (σ^2) is calculated, as shown in Eq. 5. Using the mean, variance and a set of pre-trained values (γ_i, β_i), the output image pixels are calculated, as shown in Eq. 6.

$$\mu = \frac{1}{IFM} \cdot \sum_{i=1}^{IFM} x_i \tag{4}$$

$$\sigma^2 = \frac{1}{IFM} \cdot \sum_{i=1}^{IFM} (x_i - \mu)^2 \tag{5}$$

$$y_i = \gamma_i \cdot \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta_i \tag{6}$$

A straightforward way to compute this function would be in three separate loops iterating over the 3rd dimension, nested in the loops iterating over the 1st and 2nd dimensions. With this approach only one output pixel is generated every 3 clock cycles for a parallelization degree of zero. Therefore, we created three functions to compute μ, σ^2 and y_i, which are used in a pipelined manner inside the three nested loops. As a result, the overall latency is as follows: $(ROWS \cdot COLS + 2) \cdot \frac{IFM}{V} + P$. Two times $\frac{IFM}{V}$ additional clock cycles are required, since each mini-batch must pass through these three stages in a pipeline manner. The input data of a mini-batch (B) is stored in a buffer in the first stage to be used for the next two stages. Since there are 3 stages, 3 consecutive input vectors of size IFM must be stored in buffers. The weight vectors γ and β are read in the third stage at the first use and buffered for the further use.

To calculate (μ) and (σ^2) a sum of values must be computed. Like in the convolution filter, the sum is computed using fixed-point numbers, because a floating-point sum increases the latency by a factor of 5. Therefore, floating-point numbers are converted and saturated to a 32-bit wide integer value. Since the normalization ($\frac{1}{IFM}$) is a constant, it can be pre-computed to replace the division by a multiplication. Both calculations are easy to vectorize, since only the sum needs to be parallelized. For the parallelization of the last stage which computes y_i, the term $\frac{1}{\sqrt{\sigma^2+\epsilon}}$ can be pre-computed once. This has a big impact on the resource usage when vectorizing the function, since the division and square root are the most resource consuming functions. Due to the accuracy, $\frac{1}{\sqrt{\sigma^2+\epsilon}}$ is calculated using floating-point numbers.

There are different variants of the Batch Normalization. One of them avoids the calculation of (μ) and (σ^2). In this variant, the two values are passed to the function as additional parameters, as shown in Eq. 7. Since both values are constants, the value of (c_i) can be pre-calculated after training the neural network. As a result, this variant of the Batch Normalization is very resource-efficient. The latency of the hardware function is: $ROWS \cdot COLS \cdot \frac{IFM}{V} + P$.

$$dst_i = \gamma_i \cdot (src_i - \mu_i) \cdot c_i + \beta_i, \quad c_i = \frac{1}{\sqrt{\sigma_i^2 + \epsilon}} \tag{7}$$

3.2.6 Fully Connected

The *Fully Connected* layer is an essential component of most CNNs. It is one of the last layers and is used for the final classification decision. Simplified it is a 3D convolution with a (1 × 1) kernel on an image with (1 × 1) pixel. However, the IFM and OFM can be very large. The weights, biases and input image are buffered on first use. However, since each weight/bias is read only once per image, it is recommended not to buffer them if the weight matrix becomes too large. The summation of Eq. 8 has been implemented using fixed-point numbers for the floating-point implementation. Therefore, the multiplication result is converted and saturated to a 32-bit wide number before summation and converted back afterwards. Fixed-point numbers were used, since a summation with floating-point numbers increased the total latency by a factor of 5. The fixed-point position is set by a parameter. Depending on the degree of parallelization, V multiplications are calculated in parallel and added together. After summation, the data must be shifted back due to the fixed-point multiplication according to the rounding policy. Then the bias is added. When using fixed-point values, the result is converted back to the output format according to the overflow policy. The latency of the hardware function is: $OFM \cdot \frac{IFM}{V} + P$.

$$dst_{ofm} = \sum_{ifm=1}^{IFM} (src_{ifm} \cdot weight_{ofm,ifm}) + bias_{ofm} \tag{8}$$

3.2.7 Softmax

The *Softmax* layer normalizes an input vector into a probability distribution and limits the output to a range between 0 and 1. It is used to determine the probability of several classes at once. The calculation shown in Eq. 9 is done in two parts. The first part computes the sum and stores the exponents of the inputs into a buffer. Due to the high range of values in this function all operations are done using floating-point operations. However, for the same reason as in the previous functions, the summation is calculated using fixed-point numbers. Therefore the exponent result is converted and saturated into a 32-bit fixed-point number before summation. For each element in the input vector, (V) exponents are calculated, stored and added to the summation. The second part calculates the division of

Eq. 9. For fixed-point numbers, the division result must be shifted (multiplied) to the correct position according to the rounding policy. Depending on the parallelization degree, (V) output elements are computed. The latency of the hardware function is: $2 \cdot \frac{IFM}{V} + P$.

$$dst_{ifm} = \frac{e^{src_{ifm}}}{\sum_{i=1}^{IFM}(e^{src_i})} \tag{9}$$

3.3 MobileNets Architecture

MobileNets [9] were presented by Google Inc. and were developed for mobile and embedded vision applications. MobileNets utilizes a combination of depthwise separable convolutions and pointwise convolution to form lightweight deep neural networks. These networks also introduced two global parameters for the width and resolution multiplier to define different sizes of the networks. The different networks based on these parameters have different latency and accuracy. This allows using optimum networks to match the design requirements of the system.

MobileNets architecture is based on depthwise separable convolution as mentioned before. A standard convolution can be factorized into a depthwise convolution and a pointwise convolution. Depthwise separable convolution separates filtering and combines inputs into two layers, on contrary to a standard convolution. This factorization of the convolution layer results in a reduction in model size and computation requirements of the algorithm. This concept is used in MobileNets in order to have light-weighted neural networks. The first layer of MobileNets is a full convolution layer. Later layers are a combination of depthwise convolution and pointwise convolutions. All convolutions are followed by a Batch Normalization layer and activation layer (ReLU). The final Fully Connected layer has no non-linearity and is followed by a Softmax layer. Before the final Fully Connected layer an average pooling is used to reduce the spatial resolution. In total MobileNets has 28 layers.

Figure 3 shows the hardware implementation of the different layers of MobileNets, which is parameterizable. The different modules are interconnected, with module 1 containing the first layer and module 15 the last layer, creating a very deep pipeline. The input of the first layer in module 1 is connected to a data width converter and gets its data from the global memory. To optimize the memory bandwidth, it receives an e.g. 64-bit wide input and converts it to the desired vector size (V_{IFM}) of the 3D convolution. The output vector (V_{OFM}) is then converted to the vectorization (V_{PW}) of the Batch Normalization and activation layers. All layers and conversion units of the pipeline are connected via very small FIFO buffers. They are marked by a thicker line in the figure.

The modules 2 to 14 all have the same structure with different parameter settings. The first three layers, which include a depthwise convolution, Batch Normalization and activation layer, all have the same degree of parallelization (V_{DW}). Again, the data for pointwise convolution must be converted to (V_{IFM}) and then to (V_{OFM}). The last two layers have a parallelization degree of (V_{PW}). With these 4 vectorization parameters $(V_{DW}, V_{IFM}, V_{OFM}, V_{PW})$ the optimal configuration for the desired amount of resources can be found, as shown in the evaluation. Data width converters can also be connected between the various modules. However, they were not needed in the final configuration.

Module 15 contains the last layer and its output is therefore connected to a e.g. 64-bit wide DMA via a data width converter. This module only needs the parameter (V_{DW}) for pooling and the parameter (V_{IFM}) for the input of the Fully Connected layer. The Softmax layer is not computationally intensive enough to become a bottleneck. In general, the vector parameters must be set so that no single layer becomes a bottleneck, since the slowest layer limits the speed of the others. The different modules contain scatter engines to distribute all coefficients to the local buffers. This allows all coefficients to be preloaded with optimal utilization of the memory bandwidth. The scatter engine also reduces the number of DMAs needed to access memory to load new coefficients. They require data width converters, since each local buffer has a different depth of its elements depending on the degree of parallelization of the corresponding layer. The HiFlipVX data converter can also convert between widths that are not multiples of each other. Therefore the data for the different local buffers must be aligned to the data type of the scatter engine in global memory.

3.4 High-Level Synthesis Directive Usage

In this work we use 8 different directives (`pragma HLS`): `inline`, `interface`, `data_pack`, `dataflow`, `stream`, `resource`, `pipeline`, `array_partition`. All internal and callable library functions are inlined using the `inline` directive.

The `interface` directive is only needed in wrapper functions, which instantiate the library functions and set the template parameters. There is an example test bench for each function of the library and the different MobileNets layers in the main file. When using Xilinx SDSoC no `interface` directive is be needed. For the SDSoC tool we set the `ap_fifo` protocol for all ports. For Xilinx Vivado HLS we set the AXI4-Stream (`axis`) protocol as interface for the ports. It is a simple handshaking protocol most Xilinx IP-Cores use. Additionally, we deactivate the control port of all IP-Cores in Vivado HLS

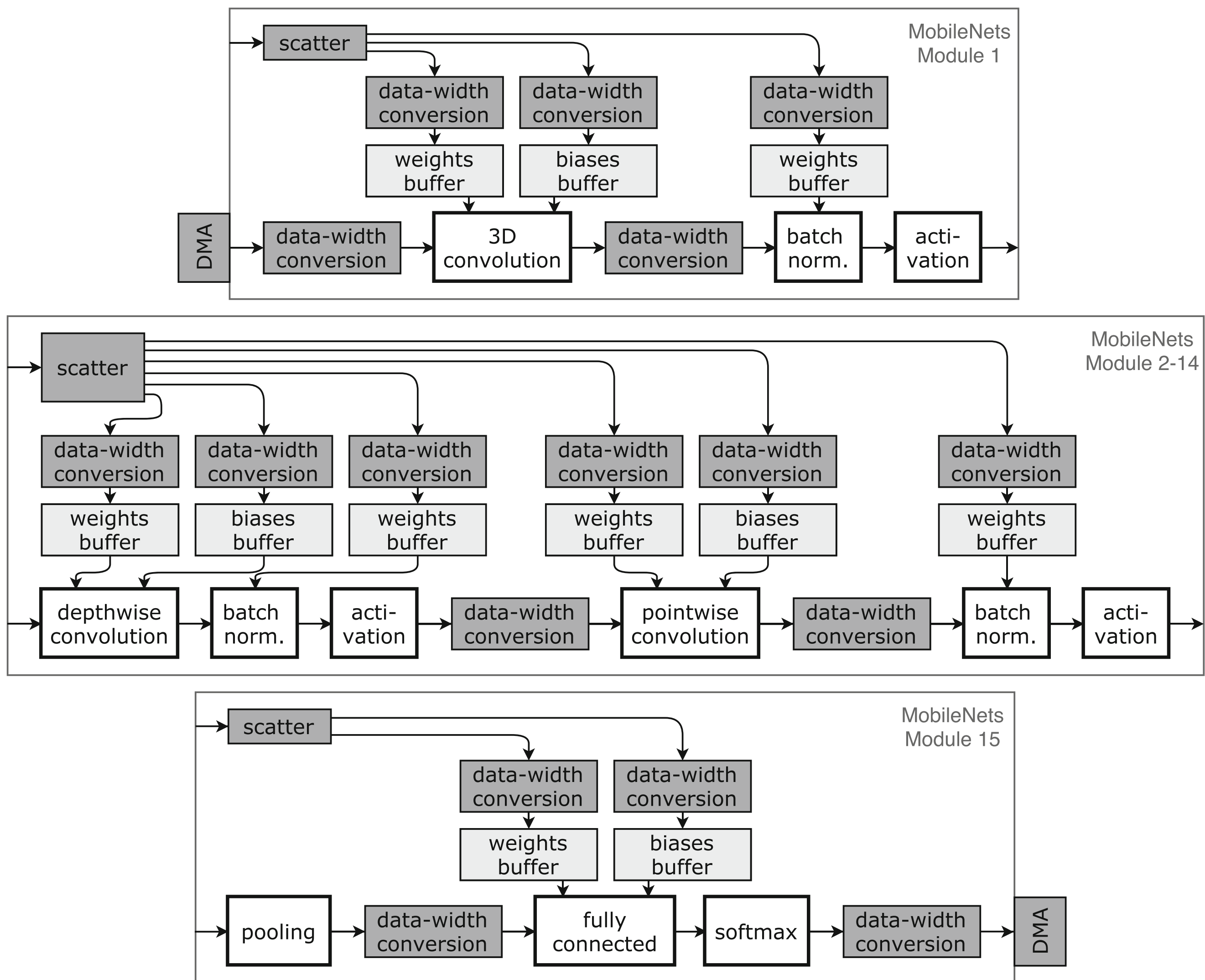

Figure 3 Block design of the MobileNets hardware implementation. MobileNets has been separated into 15 modules. The modules are directly connected to each other in the order of their numbering. Local buffers are marked in light gray. Data movers blocks are marked in dark gray. Multiple scatter units can be connected to the same DMA.

(ap_ctrl_none port=return). This port should not be deactivated for SDSoC. The (__SDSCC__) macro is globally set by the SDSoC tool and is used by our library to automatically switch between the two Xilinx tools. Setting the ap_fifo ports and using the C99 style for arrays, our library does not need any specific SDSoC directives (pragma SDS). As mentioned in Section 3.2, we use the (vx_image_data<DATA_TYPE, VEC_SIZE>) data type for the function ports, to apply vectorization and set the *last* and *user* bits of the AXI4-Stream protocol if needed. To achieve the full bandwidth, all callable library functions use the data_pack directive for their ports. Additionally, we use the data_pack directive for our internal buffers and FIFOs, to reduce the fragmentation of the utilized (BRAMs).

The dataflow directive enable task-level pipelining. It is needed to create the streaming applications in the three different MobileNets layers shown in Figure 3. To enable streaming between the different functions of the MobileNets layers, FIFOs are needed. Therefore, the stream directive is used for these FIFOs using a depth of 8. The small depth allows to use LUTs instead of BRAMs for the FIFOs, since BRAM is often a limiting resource. Within all library functions we use the pipeline directive with the goal to achieve an *initiation interval* of one. Since all loops below the pipeline directive are unrolled automatically, there is no need of using the unroll directive.

Therefore, the resource directive is set to FIFO_LUTRAM for these FIFOs. For most internal buffers, shown in Figures 1 and 2, we set the resource directive to

use LUTs (RAM_2P_LUTRAM) or BRAMs (RAM_2P_BRAM) depending on their size. RAM_2P_BRAM has been used for most weight and line buffers. RAM_2P_LUTRAM has been used for most bias, window and input buffers. The use of the resource directives should be used with caution, since it can also have a negative effect. In most cases it is advisable to give the tool the choice, because then it can select according to the total resource usage, bit-widths and selected frequency. The array_partition directive is needed if the LUT and BRAM memories do not provide the required bandwidth. E.g. to separate the window buffers of Figure 1. The array_partition directive is also used quite often to completely partition C++ arrays into registers, like for the white boxes in Figure 1.

4 Evaluation

In this section a detailed evaluation of the different functions of the library is made. Different parameter settings are evaluated to make general assumptions. Furthermore, designing larger algorithms is evaluated using MobileNets.

4.1 Single Layers

This part evaluates the different layers of the proposed library. We tested the design on a ZCU104 MPSoC FPGA from Xilinx using the 2019.1 tool chain including SDSoC and Vivado HLS. To obtain the implementation results, we built a design with SDSoC and took the results of the single functions from the Vivado project. All functions in the library have several parameters which can be changed at compile time. Table 2 shows the default configuration of the parameters of the different layers tested in this evaluation. The table also shows the high configurability of the library.

On the left side of the table are the normal parameters of a neural network, which are also needed in non-FPGA designs. Additionally we support 2 pooling types and 9 activation function types. In our terminology, batches are images that are processed one after the other. Increasing this parameter has the advantage that a function can read in pixels of a new image before it has finished the calculation of the last image. Coefficients can be buffered on first use, which does not need to be repeated for the other input images (batches). The batch size can be set for all functions in the library. The input resolution can differ from the output resolution, but has to be bigger. This is only possible for the two convolution functions and the pooling function to implement a stride. Only the 3D convolution and the Fully Connected layers have both (IFM) and (OFM), all other functions only have one feature map (IFM). For resolution, batch amount and the feature map size we allow a value between 1 and 2028. The bias size can be (0), (OFM) or $(BATCHES{\cdot}OFM)$ for the two convolution functions and the Fully Connected layer. The kernel size can be changed for the two convolution functions and is $(n \times m)$, where (n) and (m) can be different but must be odd numbers and must be in the range of 1 and 9. It is the same for the pooling size, but the numbers can also be even. Pooling and padding sizes can only be set for the pooling function. The padding size can be between 0 and the half of the pooling size. The convolution functions automatically use a padding, which is the half of the kernel size $\left(\left\lfloor \frac{K_{y,x}}{2} \right\rfloor\right)$.

On the right side of the table there are parameters that are more specific to the FPGA design, such as frequency changes. The (V_{IFM}) parallelization is used in all functions. The (V_{OFM}) parallelization is only needed for 3D convolution and Fully Connected layers, for exploration and to further improve the performance. For both parallelization parameters we allow a value between 1 and 128. We allow different data types for the inputs/outputs and weights of the different layers (int8, uin8, int16, uint16, float32). The biases can have a different data type if fixed-point numbers are used (int8, uin8, int16, uint16, int32, uint32, float32). This approach has been suggested by some CNN algorithm implementations. The fixed point position determines the size of the fraction and must be below the number of digits of the data type. For signed data types, at least 1 bit is required for the integer part. For arithmetic calculations, mainly for fixed-point numbers, we must check for overflow and perform the wanted rounding policy. If an overflow occurs data can either be truncated or saturated to its maximum/minimum value. For fixed-point arithmetic operations, the data can be rounded to zero or the nearest number. Coefficients (weights and biases) can be buffered during execution within the function (buffer coefficients). In Figure 3, this is done outside the function to increase efficiency of the coefficient reading process.

To verify the correctness of the library functions, we calculate the mean absolute percentage error (MAPE) of

Table 2 Default configurations for the changeable parameters of the different layers.

batches	4	v_{ifm}	1
input	64x64	v_{ofm}	1
output	64x64	frequency	100 Mhz
IFM	32	data type	uint8
OFM	32	bias data type	uint8
bias size	OFM	fixed point position	8
kernel size	3x3	overflow	saturate
pooling size	2x2	rounding	to zero
padding size	1x1	buffer coefficients	yes

Table 3 MAPE (mean absolute percentage error) between the 32-bit floating point baseline software implementation and the various hardware implementations.

layers	uint16	int16	float32
3D convolution	0.3413	0.6804	0.00003
depthwise conv.	0.0127	0.0261	0.00000
pooling (max)	0.0000	0.0000	0.00000
activation (relu)	0.0000	0.0000	0.00000
batch normalization	0.0390	0.1012	0.00004
fully connected	0.0000	0.3421	0.00000
softmax	0.2104	0.4245	0.00001

our hardware implementation compared to a floating-point baseline one. Table 3 shows the results using the default configuration and quantized random input numbers in the range between 0 and 1, where $\{x \in \mathbb{R} | 0 \leq x < 1\}$. The calculation of MAPE has a problem if the divisor is zero. Therefore we do not consider results where the divisor is less than 10^{-6}. The fixed-point positions for the data type in the table are 16 (uint16), 15 (int16) and 24 (float32). The MAPE of 0.68% for the 3D convolution is due to the high number of multiplications and additions for each output pixel. A similar behavior can be observed with the other functions, where many variables have to be added and/or multiplied together. The float32 computation can have a very small error for the functions that have to calculate a sum over several loops, because we had to use fixed point arithmetic for this summation. Of course, if numbers had to be saturated, the MAPE would be higher, but this was not meant to be proven by this approach, since it is generally the case for fixed point numbers.

Table 4 shows the resource utilization of the implemented and synthesized (grey) designs using the default configuration. In this table, the Softmax and Fully Connected layers have 256 IFM and 256 OFM, since the resolution for these layers is (1×1). As it can be seen from the table, the difference between the estimated synthesis results from SDSoC and the implemented results from Vivado is quite large for FFs and LUTs. The implementation results in the table do not include the additional blocks that SDSoC integrates into the HW design. The DSP behavior is different because we let the tool decide whether to use LUTs or DSPs for the arithmetic calculation, as this can vary depending on the application. The Fully Connected layer usually has many coefficients and therefore requires a lot of BRAM. Therefore, it may make sense not to buffer the weights, since each weight is only required once per batch. The 3D convolution consumes more BRAM than the depthwise convolution because it has to buffer more coefficients.

In addition, the table shows the estimated latency per batch. As it is well known, the process of 3D convolution is the most computationally intensive part in many CNN algorithms and must therefore be parallelized more. The Softmax function is the least computationally intensive function and could therefore be executed on a CPU in a HW/SW co-design, as this function is also quite resource intensive. By adding the proposed multi-stage pipelining approach, Batch Normalization can calculate the three internal functions almost in the same time as the activation layer. Due to this approach and the computationally intensive operations like division and square root, more resources are needed. Depthwise convolution and pooling require some additional cycles due to the line buffers.

Table 5 shows the resource usage of the implemented design from the various activation functions using unsigned 16-bit data types. As expected, all functions that include an exponent, logarithm, or division in their equation consume more resources. Using exponential functions instead of the hyperbolic functions could reduce resource usage. For the square root function, there is an option for relaxed mathematical calculation to reduce resource usage by reducing the precision of the fraction part. The difference in accuracy can be seen with a MAPE of 0.37 %. Due to the accuracy, mainly floating point operations were used for the computational-intensive functions. However, due to quantization, there is still a small error rate left for these functions.

Table 4 Resource utilization and latency per batch of implemented (black) and synthesized (grey) designs.

	BRAM		DSP		FF		LUT		latency
3D convolution	3	3	5	9	1293	439	2114	534	4326401
depthwise conv.	2	2	5	9	872	233	1488	407	135201
pooling (max)	0.5	0.5	0	0	162	128	690	191	135201
activation (relu)	0	0	0	0	26	26	118	17	131073
batch normalization	0	0	14	13	6618	3290	6716	3532	131102
fully connected	17	17	0	1	421	347	842	282	65537
softmax	0	0	19	19	2568	1987	4610	2939	522

Fully Connected and Softmax layers have 256 IFM and 256 OFM

Table 5 MAPE (mean absolute percentage error) and resource utilization of the implemented design of the various activation functions using unsigned 16-bit data types.

	BRAM	DSP	FF	LUT	MAPE
logistic	0	15	1362	2146	0.00126
hyperbolic	0	17	1549	2370	0.02543
relu	0	0	26	17	0.00000
brelu	0	0	26	25	0.00000
softrelu	0	28	1418	2112	0.00144
abs	0	0	26	17	0.00000
square	0	1	28	28	0.00000
sqrt	0	0	164	384	0.00139
sqrt (relaxed)	0	0	113	243	0.36990
linear	0	0	26	25	0.00000

Figure 4 shows the relative resource usage for various parameter settings compared to the default configuration. As expected, a change in frequency mainly increases the FFs (43% on average), but also the LUTs (8% on average). However, it has no effect on the BRAMs or DSPs. For designs with higher accuracy or for fast integration and testing, the library also supports floating point numbers. They have no effect on the latency of the various library functions, except for additional pipeline stages, but have a high impact on resource utilization: +432% LUTs, +784% FFs and +423% DSPs. When using 16-bit fixed-point values to increase accuracy, there is only a small increase for LUTs (25%), FFs (17%) and DSPs (2%). This again shows the importance of quantization in FPGAs. The BRAM usage always scales with the bit width of the data type used. Increasing the kernel size has a similar effect for 3D and depthwise convolution. In both cases the DSPs grow with the kernel size. The BRAM increase depends on the coefficient size ($k_y \times k_x$) and the line buffer amount of ($k_y - 1$). LUTs and FFs are only increased by 85% and 65% respectively for 2.78× the amount of weights.

A more detailed investigation of parallelization was done, because finding the right parameters is important for an efficient and performant design. The Batch Normalization

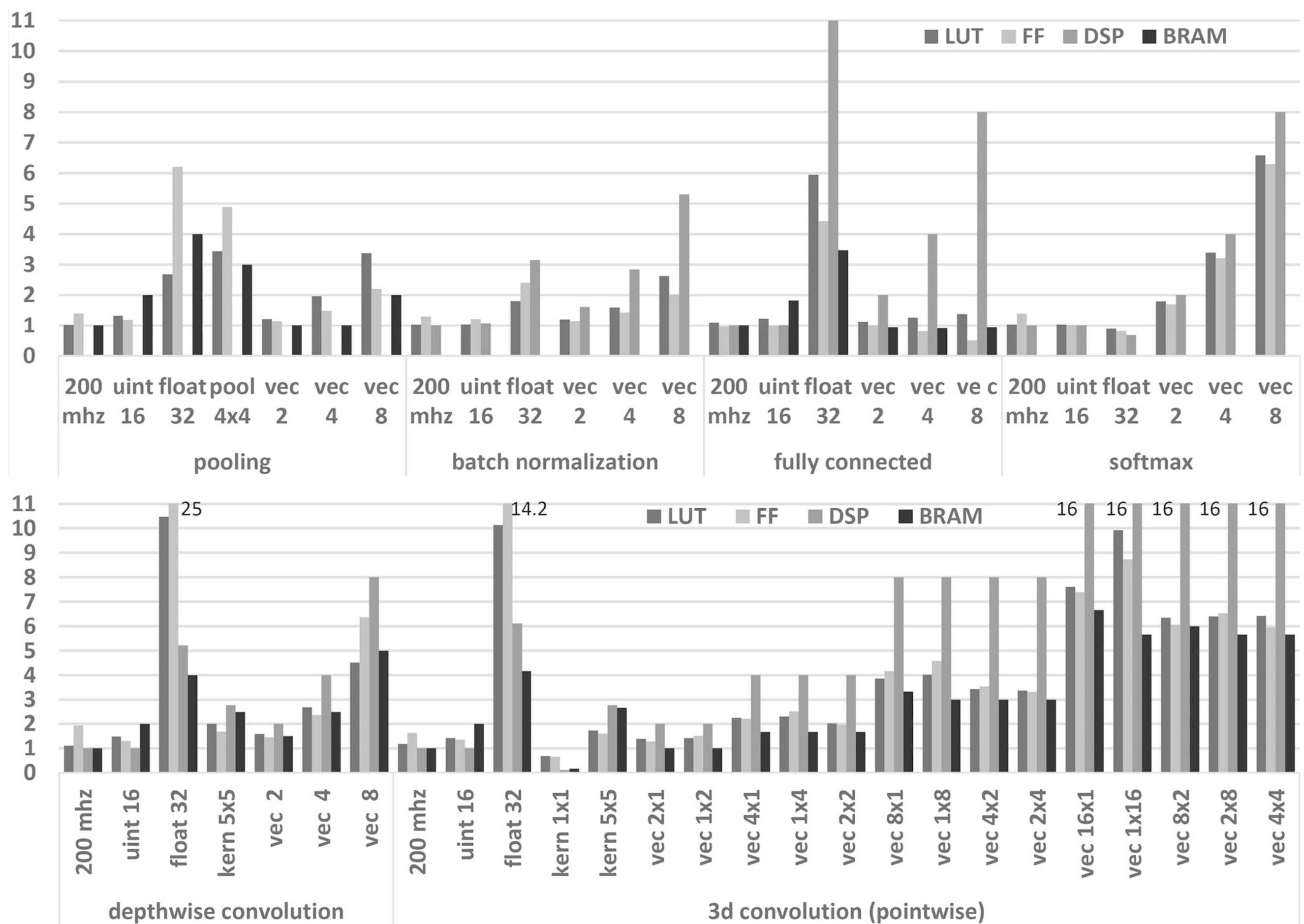

Figure 4 Relative resource utilization for various settings compared to the default configuration. Value is not reported if it is zero. 3D convolution has a vectorization of $v_{ifm} \times v_{ofm}$.

layer scales well with parallelization because resource-intensive functions do not need to be calculated multiple times as described in Section 3.2.5. Only the increase in DSPs approximates to a linear behavior. The DSPs of all other functions scale linearly with the degree of parallelization. The LUTs and FFs of the Pooling layer scale less than linearly with the degree of parallelization. The Fully Connected layer even shows a reduction of FFs and BRAMs due to fragmentation. The 3D convolution has a combined vectorization of ($V_{IFM} \times V_{OFM}$). Different combinations of (V_{IFM}) and (V_{OFM}) were tested to find an optimized combination. The combined vectorization results in a parallelization (V) for 2 ($1 \times 2|2 \times 1$), 4 ($4 \times 1|4 \times 4|2 \times 2$), 8 ($8 \times 1|1 \times 8|4 \times 2|2 \times 4$) or 16 ($16 \times 1|1 \times 16|8 \times 2|2 \times 8|4 \times 4$). Some assumptions can be made when comparing these combinations. The greater the imbalance between (V_{IFM}) and (V_{OFM}), the more resources are used on average. If, for the same (V), (V_{IFM}) is greater than (V_{OFM}), the average usage of LUTs and FFs increases slightly by 6% and 10% respectively. On the contrary, a high (V_{IFM}) can cause more BRAM to be used if it worsens line buffer fragmentation.

Additionally, one 3D convolution layer has been implemented with a high parallelization to show the performance improvement in comparison to a baseline implementation, which is running on the ARM processor of the ZCU104 MPSoC at a frequency of 1.2 GHz in release mode using the O3 optimization option. Using the same default configuration with ($V_{OFM} = 8$), ($V_{IFM} = 8$) and a frequency of 200 MHz, an acceleration of 260 was achieved when the convolution function was executed on the real system using SDSoC. The measurements were performed with the ARM processor, on which no operating system is running. The consumed resources for the convolution function are: 8858 LUTs, 7679 FFs, 576 DSPs and 66 BRAMs. The BRAM has increased due to fragmentation and a high demand of on-chip bandwidth. The execution time of the hardware is $837\mu s$, which includes the cache flushing and data movement between the FPGA and DMA.

4.2 MobileNets

Before implementing the MobileNets layers onto hardware, the optimal parameters must be set. When creating a deep pipeline, the system normally is as fast as its slowest component. Table 6 shows our offline calculations for an optimal setting of the different modules containing the MobileNets layers. All parameters, which are not reported in the table use the default configuration. The parameter values for the resolution and feature maps are set by the algorithm. Using the latency equations of Section 4.1, the estimated latency can be calculated. The number of pipeline stages was ignored in this estimation as it has almost no impact. The maximum latency in the right column shows the bottleneck of the design. In the next step, the vectorization (V) settings discussed in Section 3.3 are adapted to improve

Table 6 Shows proposed vectorization (V) setting for MobileNets layers of Section 3.3.

	Resolution				Parallelization				Estimated Latency (clock cycles)				
	$input$	$output$	IFM	OFM	v_{dw}	v_{ifm}	v_{ofm}	v_{pw}	dw_{conv}	dw_{bn}	pw_{conv}	pw_{bn}	max
1	224x224	112x112	3	16	–	3	8	2			101250	100368	101250
2	112x112	112x112	16	32	2	8	8	4	102152	100368	100352	100368	102152
3	112x112	56x56	32	64	4	8	8	2	102152	25104	100352	100416	102152
4	56x56	56x56	64	64	2	8	16	2	103968	100416	100352	100416	103968
5	56x56	28x28	64	128	2	8	8	1	103968	25152	100352	100608	103968
6	28x28	28x28	128	128	1	8	16	1	107648	100608	100352	100608	107648
7	28x28	14x14	128	256	1	8	8	1	107648	25344	100352	50688	107648
8	14x14	14x14	256	256	1	8	16	1	57600	50688	100352	50688	100352
9	14x14	14x14	256	256	1	8	16	1	57600	50688	100352	50688	100352
10	14x14	14x14	256	256	1	8	16	1	57600	50688	100352	50688	100352
11	14x14	14x14	256	256	1	8	16	1	57600	50688	100352	50688	100352
12	14x14	14x14	256	256	1	8	16	1	57600	50688	100352	50688	100352
13	14x14	7x7	256	512	1	8	8	1	57600	13056	100352	26112	100352
14	7x7	7x7	512	512	1	8	16	1	32768	26112	100352	26112	100352
15	7x7	1x1	512	1000	1	8	1	1	25088		64000	2000	64000

Latency is calculated for functions in Figure 3 separately without pipeline stages. Depthwise (dw) & pointwise (pw) latency of Batch Normalization (bn) & convolution are reported. Maximum latency of all functions within a layer is shown on the right

Table 7 Final results of the three MobileNets modules shown in Section 3.3 executed separately on the ZCU104.

	module 1	module 2	module 15
ARM (ms)	34.166	53.221	9.944
FPGA (ms)	0.966	0.902	0.489
speed-up	35.4	59.0	20.3
LUT	11881	16914	10579
FF	13265	16660	5773
DSPs	237	140	27
BRAM	1	20	263.5

The FPGA runs at 200 MHz

the max latency, while keeping the available resources for DSPs and BRAMs into account. Since these two resources can be easily estimated and are in most cases the limiting resources for CNNs. The activation layer is not taken into account, since it has the same parallelization as the Batch Normalization, but a slightly lower latency. In the table: (v_{dw}) refers to (dw_{conv}) and (dw_{bn}); (v_{ifm}) and (v_{ofm}) refer to (pw_{conv}); (v_{pw}) refers to (pw_{bn}). For module 15, (dw_{conv}) refers to the Pooling layer and (pw_{bn}) to the Softmax layer.

Table 7 shows the final implemented design executed on the ZCU104 MPSoC in baremetal. A baseline software implementation uses 32-bit floating point numbers and runs on the ARM processor at a frequency of 1.2 GHz in release mode using the O3 optimization option. Our proposed implementation uses 8-bit unsigned numbers and runs on the FPGA at a frequency of 0.2 GHz. The time measurements have been done using the ARM processor. A good speed-up has been achieved for the single modules. Module 2 has the highest speed-up, since it has the highest parallelization degree and contains most functions executed in a streaming manner. For module 1 and 2 also a frequency of 300 MHz was possible. When combining all modules to a very deep pipeline this speed-up would be even higher. When comparing the FPGAs computation time with the estimated time of the slowest function in Table 6, there is some overhead for streaming multiple functions in a pipeline, data moving between the DDR and cache flushing. This overhead is 90.8%, 76.6% and 52.9% for the modules 1, 2 and 15. When executing the layers sequentially, these numbers would be higher. To verify the propagation of the error, the MAPE value was computed for 16-bit unsigned fixed-point numbers. It was 0.21%, 0.79% and 0.78% for the modules 1, 2 and 15. The resources listed in the table contain only the modules and no DMAs. When considering the ZCU104 the resource usage is sufficient to fit all layers. For this case, the Ultra Rams would be needed and the Fully Connected layer in module 15 should not buffer its weights.

4.3 Comparisons to Related Work

Hassan et al. [8] presented a HW/SW co-design implementation of AlexNet on an FPGA. They performed the first layer of AlexNet on hardware and achieved 2147483647 clock cycles, which would be approximately 10.7 ms when considering a frequency of 0.2 GHz. For comparison, a similar convolution layer was implemented using our library with the same frequency, same parameters and 8-bit unsigned integer data types. The implemented convolution layer had a latency of 3.31 ms, which is a speed-up of 3.23. For the same layer, our work shows almost 73% less BRAM usage, demonstrating the proposed library's ability to reduce the memory consumption of large neural networks on FPGAs.

Liu et al. [19] proposed and developed a CNN accelerator for the Xilinx ZYNQ-7100 platform. They implemented the SSD-MobileNets-V1 [31] layers as test application for their proposed work. The proposed work is also HLS based and uses Vivado HLS 2016.4. We implemented the most time consuming SSD-MobileNets-V1 layers and compared them with the work of Liu et al. in Table 8. For our hardware and software implementations we used the Zynq ZCU102. For measurements we executed the algorithms on the board and measured them from the ARM processor. We show the CPU and FPGA results of our work and of Liu et al. [19]. Both implementations run at 100 MHz, to have a fair comparison, but higher frequencies can be achieved

Table 8 Comparison with a state-of-the-art implementation of computational intensive layers of the SSD-MobileNets-V1 algorithm.

	Layer 1	Layer 7	Layer 27	Layer 29
CPU [19]	2000.00	9000.00	5500.00	11000.00
Accelerator [19]	10.00	30.00	55.00	110.00
ARM Cortex A-53	82.89	339.41	377.65	82.64
Proposed 1 ($V_{IFM} \times V_{OFM}$)	(3x2) 11.38	(8x8) 6.08	(8x8) 9.52	(2x4) 11.45
Proposed 2 ($V_{IFM} \times V_{OFM}$)	(3x4) 5.95	(8x16) 3.38	(8x16) 4.92	(4x4) 5.89

All results are in ms. V_{IFM}: Parallelization of the Input Feature Map. V_{OFM}: Parallelization of the Output Feature Map

with our implementation. The table shows the execution times for different parallelization settings for IFM and OFM of our implementation. The `Proposed 2` settings should be the maximum possible in terms of available resources, if the complete algorithm is ported to the ZCU102 and the different functions stream their results between each other. If we compare the results of Layer 27 and Layer 29 of the SSD-Mobilenet-V1 network, our execution time is 11.2x and 18.7x times faster. When computing the complete streaming network of SSD-Mobilenet-V1, Layers 1 and 27 would be the bottleneck. This is intended, since layers 1, 27, 29, 31 and 33 of SSD-Mobilenet-V1 are the only layers with a 3×3 convolution kernel. Therefore, we used these layers as roofline, since they consume more DSPs than the other layers. For our work, we propose to use a quantization technique, like the TensorFlow post-training quantization [1]. This will allow us to use smaller parameters thus saving resources and power. For our work, we used unsigned 8-bit integers as data types for inputs, outputs, weights and biases. Wu et al. [35] investigated the mathematical aspect of quantization parameters on different neural networks. Also, an 8-bit quantization workflow is presented where an accuracy within 1% of the floating-point baseline is maintained. Therefore quantized parameters with smaller bit widths should be used instead of floating-point parameters. As it maintains a reasonable accuracy, achieves a higher speedup and saves resources.

5 Conclusion

In this work we have shown an HLS FPGA library for neural networks. It contains 7 different streaming capable functions to create large neural networks with deep pipelines. Due to the high parameterization of its functions the library is suitable for embedded and HPC systems. The integration in HiFlipVX allows the use of further image processing functions. Although the library was optimized by Xilinx HLS directives, it was implemented in a way that it is vendor independent. The different parameter settings and parallelization possibilities were investigated in the evaluation to make conclusions for the user. The evaluation also shows the low error rate, high performance, scalability and resource efficiency of the library. Using the MobileNets algorithm we show how to efficiently create and optimize larger designs. An efficient approach to transfer coefficients and a way to find the optimal vectorization parameters were shown. In future, we plan to enhance the library to a framework, which uses the OpenVX graph-based approach.

Acknowledgements This work has been funded partially by the German Federal Ministry of Education and Research BMBF as part of the PARIS project under grant agreement number 16ES0657 and partially by COllective Research NETworking (CORNET) project AITIA: Embedded AI Techniques for Industrial Applications. CORNET-AITIA is funded by the BMWi (Federal Ministry for Economic Affairs and Energy) under the IGF-project number: 249 EBG.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommonshorg/licenses/by/4.0/.

References

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)* (pp. 265–283). https://github.com/tensorflow/models/blob/master/research/slim/nets/mobilenet_v1.md.
2. Akgün, G., Kalms, L., Göhringer, D. (2020). Resource efficient dynamic voltage and frequency scaling on xilinx fpgas. In *International symposium on applied reconfigurable computing (ARC)* (pp. 178–192).
3. Chen, Y., He, J., Zhang, X., Hao, C., Chen, D. (2019). Cloud-dnn: an open framework for mapping dnn models to cloud fpgas. In *Proceedings of the international symposium on field-programmable gate arrays (FPGA)* (pp. 73–82). https://doi.org/10.1145/3289602.3293915.
4. Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., Temam, O. (2014). Dadiannao: a machine-learning supercomputer. In *47th annual IEEE/ACM international symposium on microarchitecture* (pp. 609–622).
5. Giduthuri, R., & Pulli, K. (2016). Openvx: A framework for accelerating computer vision. In *SIGGRAPH ASIA 2016 Courses* (pp. 14:1–14:50). https://doi.org/10.1145/2988458.2988513.
6. Guan, Y., Liang, H., Xu, N., Wang, W., Shi, S., Chen, X., Sun, G., Zhang, W., Cong, J. (2017). Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. In *25th annual international symposium on field-programmable custom computing machines (FCCM)* (pp. 152–159).
7. Guo, K., Sui, L., Qiu, J., Yu, J., Wang, J., Yao, S., Han, S., Wang, Y., Yang, H. (2018). Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *37*(1), 35–47.
8. Hassan, R., & Mostafa, H. (2020). Implementation of deep neural networks on fpga-cpu platform using xilinx sdsoc Analog Integrated Circuits and Signal Processing. https://doi.org/10.1007/s10470-020-01638-5.
9. Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H. (2017). Mobilenets:

Efficient convolutional neural networks for mobile vision applications. arXiv:1704.04861.
10. Intel (2020). Intel FPGA SDK for OpenCL Pro Edition: Programming Guide 19.4.
11. Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. arXiv:1502.03167.
12. Ji, S., Xu, W., Yang, M., Yu, K. (2013). 3d convolutional neural networks for human action recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *35*(1), 221–231.
13. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on multimedia* (pp. 675–678).
14. Kalms, L., & Göhringer, D. (2017). Exploration of opencl for fpgas using sdaccel and comparison to gpus and multicore cpus. In *27th international conference on field programmable logic and applications (FPL)* (pp. 1–4). https://doi.org/10.23919/FPL.2017.8056847.
15. Kalms, L., & Göhringer, D. (2020). *Accelerated high-level synthesis feature detection for FPGAs using HiFlipVX, chap. 7*, (pp. 115–135). New York: Springer.
16. Kalms, L., & Göhringer, D. (2020). Hiflipvx: Open source high-level synthesis fpga library for image processing. https://github.com/TUD-ADS/HiFlipVX.
17. Kalms, L., Podlubne, A., Göhringer, D. (2019). Hiflipvx: An open source high-level synthesis fpga library for image processing. In *Applied reconfigurable computing* (pp. 149–164).
18. Krizhevsky, A., Sutskever, I., Hinton, G.E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90. https://doi.org/10.1145/3065386.
19. Liu, B., Zou, D., Feng, L., Feng, S., Fu, P., Li, J. (2019). An fpga-based cnn accelerator integrating depthwise separable convolution. *Electronics*, *8*, 281.
20. Liu, Z., Chow, P., Xu, J., Jiang, J., Dou, Y., Zhou, J. (2019). A uniform architecture design for accelerating 2d and 3d cnns on fpgas. *Electronics*, *8*, 65.
21. Long, J., Shelhamer, E., Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In *2015 IEEE Conference on computer vision and pattern recognition (CVPR)* (pp. 3431–3440).
22. Omidian, H., & Lemieux, G.G.F. (2018). Janus: A compilation system for balancing parallelism and performance in openvx. Journal of Physics: Conference Series (JPCS) 012–011. https://doi.org/10.1088/1742-6596/1004/1/012011.
23. Pouyanfar, S., Sadiq, S., Yan, Y., Tian, H., Tao, Y., Reyes, M.P., Shyu, M.L., Chen, S.C., Iyengar, S.S. (2018). A survey on deep learning: Algorithms, techniques, and applications. ACM Comput. Surv 51(5). https://doi.org/10.1145/3234150.
24. Qasaimeh, M., Denolf, K., Lo, J., Vissers, K., Zambreno, J., Jones, P.H. (2019). Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels. In *International conference on embedded software and systems (ICESS)* (pp. 1–8).
25. Ren, S., He, K., Girshick, R., Sun, J. (2017). Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *39*(6), 1137–1149.
26. Sekar, C. (2017). Hemasunder: Tutorial t7: Designing with xilinx sdsoc. In *30th international conference on VLSI design and 16th international conference on embedded systems (VLSID)* (pp. xl–xli). https://doi.org/10.1109/VLSID.2017.97.
27. Song, L., Wang, Y., Han, Y., Zhao, X., Liu, B., Li, X. (2016). C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization. In *Proceedings of the 53rd Annual Design Automation Conference (DAC)*. https://doi.org/10.1145/2897937.2897995.
28. Suda, N., Chandra, V., Dasika, G., Mohanty, A., Ma, Y., Vrudhula, S., Seo, J.S., Cao, Y. (2016). Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Inproceedings of international symposium on field-programmable gate arrays (FPGA)* (pp. 16–25). https://doi.org/10.1145/2847263.2847276.
29. Taheri, S., Behnam, P., Bozorgzadeh, E., Veidenbaum, A., Nicolau, A. (2019). Affix: Automatic acceleration framework for fpga implementation of openvx vision algorithms. In *International symposium on field-programmable gate arrays (FPGA)* (pp. 252–261). https://doi.org/10.1145/3289602.3293907.
30. Tapiador Morales, R., Rios-Navarro, A., Linares-Barranco, A., Kim, M., Kadetotad, D., Seo, J.S. (2016). Comprehensive evaluation of opencl-based convolutional neural network accelerators in xilinx and altera fpgas coRR.
31. Tensorflow (2020). Ssd mobilenet v1. https://tensorflow.org/lite/models/object_detection/overview.
32. Venieris, S.I., & Bouganis, C. (2017). Latency-driven design for fpga-based convolutional neural networks. In *27Th international conference on field programmable logic and applications (FPL)* (pp. 1–8).
33. Wang, Y., Xu, J., Han, Y., Li, H., Li, X. (2016). Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family. In *53rd design automation conference (DAC)* (pp. 1–6).
34. Winterstein, F., Bayliss, S., Constantinides, G.A. (2013). High-level synthesis of dynamic data structures: A case study using vivado hls. In *International conference on field-programmable technology (FPT)* (pp. 362–365). https://doi.org/10.1109/FPT.2013.6718388.
35. Wu, H., Judd, P., Zhang, X., Isaev, M., Micikevicius, P. (2020). Integer quantization for deep learning inference: Principles and empirical evaluation.
36. Xilinx (2019). xfopencv. https://github.com/Xilinx/xfopencv.
37. Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., Cong, J. (2015). Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the international symposium on field-programmable gate arrays (FPGA)* (pp. 161–170). https://doi.org/10.1145/2684746.2689060.
38. Zhang, C., Sun, G., Fang, Z., Zhou, P., Pan, P., Cong, J. (2018). Caffeine:towards uniformed representation and acceleration for deep convolutional neural networks. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.
39. Zhang, J., Li, J., 25–34 (2017). Improving the performance of opencl-based fpga accelerator for convolutional neural network. In *Inproceedings of international symposium on field-programmable gate arrays (FPGA)*. https://doi.org/10.1145/3020078.3021698.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.